Správa o realizácii projektu

Martin Kvietok
127214
Štvrtok 11:00
doc. Ing. Ján Lang, PhD.

Obsah

Úvodné informácie

Názov projektu: idealMEET

Zámer projektu: Hlasovanie o ideálnom čase na míting.

Popis projektu:

Naplánovanie stretnutí v organizácii s mnohými členmi je často náročné a vyžaduje značné úsilie. Vyhovieť každému sa zdá byť niekedy takmer nemožné...

idealMEET je navrhnutý tak, aby tento problém vyriešil, a to automatickým vyhľadávaním a vyhodnocovaním **najvhodnejšieho času s ohľadom na všetky relevantné aspekty**.

Pri prvom použití sa prihlásite do systému a vyberiete, či chcete pokračovať v existujúcej organizácii alebo vytvoriť novú. Následne môžete dynamicky pridávať alebo odstraňovať lokácie (štáty a mestá), kde organizácia pôsobí, a rozdeliť používateľov podľa ich úlohy a postavenia v organizácii. (Prezident, Vice, Líder, Člen, Kandidát) Každý typ používateľa má svoje preferencie a práva týkajúce sa počtu hlasov.

Prvým krokom v plánovaní stretnutí je vytvorenie voľných časových slotov v určenom formáte (od-do) a s ohľadom na trvanie a kapacitu miestnosti. idealMeet automaticky vyhodnotí, či by malo ísť o osobné (OfficeMeet), online (OnlineMeet) alebo hybridné stretnutie (HybridMeet), a to na základe kapacity miestnosti a preferencií používateľov. Možnosť "Vote all" umožňuje paralelné hlasovanie všetkých pobočiek súčasne, čo je obzvlášť efektívne v prípade veľkých organizácií s viacerými pobočkami. Táto funkcia zjednodušuje a urýchľuje rozhodovací proces.

Cieľom je minimalizovať zbytočné a neefektívne stretnutia a vytvoriť optimálny harmonogram. Na konci procesu sú vybrané tri najvhodnejšie možnosti, ktoré spĺňajú stanovené kritériá a sú vhodné pre všetkých členov organizácie na danej lokalite.

Grafické rozhranie idealMeetu ponúka dynamickú interakciu s používateľom prostredníctvom samostatných tlačidiel funkcií a troch výstupných panelov, ktoré zobrazujú všetky relevantné udalosti. Pri vstupe údajov sa používa tlačidlo Enter na potvrdenie a uskutočnenie akcie. Výstupné panely umožňujú užívateľovi sledovať priebeh procesu a všetky aktivity v reálnom čase, čím sa zabezpečuje transparentnosť a efektívne riadenie stretnutí a plánovania.

// idealMEET je vyvinutý s cieľom umožniť jeho implementáciu v študentských organizáciách po celom svete, vrátane organizácií ako napríklad IAESTE.

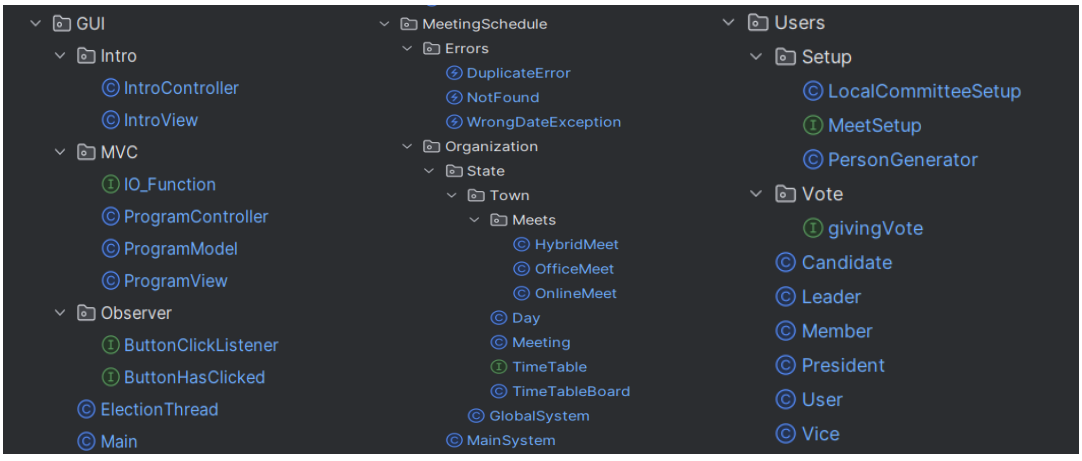
OOP princípy

Zmysluplné rozdelenie do balíkov

V projekte som štruktúroval triedy do príslušných balíkov, čím som dosiahol lepšiu organizáciu a prehľadnosť. Tieto balíky som rozdelil do troch hlavných kategórií:

1. **GUI (Grafické užívateľské rozhranie):**
   - Obsahuje triedy týkajúce sa grafického užívateľského rozhrania.
   - Implementoval som architektonický vzor Model-View-Controller (MVC) na oddelenie aplikačnej logiky od GUI.
   - Použil som aj návrhový vzor Observer na zlepšenie komunikácie medzi časťami aplikácie.
   - Okrem toho, v tomto balíku sú aj špecifické triedy ako ElectionThread, ktoré sú dôležité pre procesy súvisiace s voľbami.
2. **MeetingSchedule (Plánovanie stretnutí):**
   - Obsahuje triedy, ktoré riadia hlavnú štruktúru programu a manipuláciu s plánmi stretnutí.
   - Obsahuje aj triedy zamerané na konkrétne stretnutia (napr. triedy Meeting), čo zvyšuje jeho prehľadnosť a zjednodušuje údržbu.
3. **Users (Používatelia):**
   - Obsahuje hierarchiu používateľov a základné nastavenie komisií a volebného procesu.
   - Rozdelil som používateľov do samostatného balíka, čo zlepšuje riadenie prístupu a povolení v systéme a zvyšuje prehľadnosť.

Pri implementácii architektonického vzoru MVC som sa sústredil na oddelenie Modelu, View a Controllera, a vnútorné triedy som ďalej rozdelil do balíkov tam, kde to bolo potrebné. Týmto spôsobom som zabezpečil lepšiu organizáciu, prehľadnosť a štruktúrovanie tried v projekte.



Zapuzdrenie / Encapuslation

Vo svojom programe som uplatňoval koncept zapuzdrenia prakticky v celej jeho štruktúre. Atribúty tried som dôsledne nastavil ako „private“ , čo zabezpečilo, že neboli priamo dostupné z okolitých tried. Na manipuláciu s týmito atribútmi som vytvoril verejné GET/SET metódy. Týmto spôsobom som zabezpečil kontrolovaný a bezpečný prístup k dátam v mojej aplikácii.

```
8 usages  Martin Kvietok *
public class ProgramModel {
    18 usages
    private MainSystem myOrganization = null;
    7 usages
    private GlobalSystem visitedState = null;
    27 usages
    private TimeTableBoard visitedTown = null;


    1 usage  Martin Kvietok
    public void setMyOrganization(String name) { this.myOrganization = new MainSystem(name); }

    8 usages  Martin Kvietok
    public MainSystem getMyOrganization() { return this.myOrganization; }

    2 usages  Martin Kvietok
    public String getMyOrganizationName() { return this.myOrganization.getName(); }

    2 usages  Martin Kvietok
    public GlobalSystem getVisitedState() { return this.visitedState; }

    2 usages  Martin Kvietok
    public TimeTableBoard getVisitedTown() { return this.visitedTown; }
```

```
view.printAllVotes(model.getMyOrganization());
```

Agregácia

V projekte dochádza k agregáciám v triedach MainSystem, GlobalSystem, TimeTableBoard, Day a User. Tieto triedy zahrňujú iné triedy, ktoré sú vytvárané alebo odstraňované nezávisle od hlavnej triedy.

Napríklad, v triede User sa agreguje trieda TimeTableBoard, ktorá je vytvorená nezávisle od triedy User. To znamená, že trieda User obsahuje referencie na objekty typu TimeTableBoard, a aj po odstránení triedy User budú tieto objekty stále existovať. Podobne platí aj v ostatných prípadoch, kde dochádza k agregácii polí tried.

Na druhej strane, v prípade triedy LocalCommitteeSetup a rovnako v MVC (ProgramController a IntroController) ide o príklad kompozície, LocalCommitteeSetup committee = new LocalCommitteeSetup();, kde LocalCommitteeSetup je súčasťou TimeTableBoard a je vytvorená v rámci tejto triedy. Komisia vytvorená týmto spôsobom je tesne spojená s triedou TimeTableBoard a jej životný cyklus je úzko viazaný na životný cyklus tejto triedy.

Trieda User (agreguje triedu TimeTableBoard)

```
2 usages
protected TimeTableBoard MyWork;
2 usages
```

Trieda TimeTableBoard (agreguje triedu User a kompozituje triedu LocalCommitteSetup)

```
public class TimeTableBoard implements TimeTable, Serializable {
    4 usages
    private ArrayList<User> users = new ArrayList<>();
    4 usages
    private ArrayList<String> ActiveVoters = new ArrayList<>();
    13 usages
    private ArrayList<Day> idealDay = new ArrayList<>();
    5 usages
    private LocalDate StartMeet;
    4 usages
    private LocalDate EndMeet;
    3 usages
    private int duration;
    3 usages
    private int capacity;
    public String Town;
    2 usages
    protected int max_index = Integer.MAX_VALUE;
    2 usages
    LocalCommitteeSetup committee = new LocalCommitteeSetup();
```

Dedenia a rozhrania

Vo svojom projekte som použil 2 vlastné dedenia (Jednoduché a Hierarchické) a implementoval som 6 rozhraní.

1: Použitie jednoduchého dedenia od vstavaných (built-in) tried. (napr. pri vlastnej výnimke, GUI alebo multithreadingu)

```
public class DuplicateError extends Exception {
```

```
public class NotFound extends Exception {
```

```
public class WrongDateException extends Exception {
```

```
public class Main extends Application {
```

```
public class ElectionThread extends Thread {
```

2: Vlastné hierarchické dedenie - trieda User implementuje rozhranie givingVote a má 5 podtried hierarchie.

```
public interface givingVote {
```

```
5 inheritors   Martin Kvietok
public class User implements givingVote, Serializable {
```

```
1 usage   Martin Kvietok
public class President extends User {
```

```
public class Vice extends User {
```

```
public class Leader extends User {
```

```
public class Member extends User {
```

```
public class Candidate extends User {
```

3: Implementácia rozhraní

```
public interface MeetSetup {
```

```
public class LocalCommitteeSetup implements MeetSetup, Serializable {
```

```
public interface TimeTable {
```

```
public class TimeTableBoard implements TimeTable, Serializable {
```

4: Kombinácia jednoduchého (Day<-Meeting) **a hieratického** (Meeting<-Online,Offline,Hybrid) dedenia.

```
public class Day implements Serializable {
```

```
public class Meeting extends Day {
```

```
public class HybridMeet extends Meeting {
```

```
public class OfficeMeet extends Meeting
```

```
public class OnlineMeet extends Meeting {
```

5: Implementácia rozhraní aj na základe použitia návrhového vzoru Observer.

```
interface IO_Function {
```

```
public interface ButtonHasClicked {
```

```
public interface ButtonClickListener {
```

```
public class IntroController implements ButtonClickListener {
public class ProgramController implements ButtonClickListener {
```

```
public class ProgramView implements ButtonHasClicked, IO_Function {
 public class IntroView implements ButtonHasClicked {
```

Polymorfizmus

V mojom kóde uplatňujem polymorfizmus viacerými spôsobmi:

1. **Preťažovanie metód:** Používam preťažovanie metód, čo znamená, že v jednej triede môžem mať viacero metód s rovnakým názvom, ale s rôznymi parametrami. Týmto spôsobom môžem mať rôzne varianty tej istej operácie, ktoré sa vykonajú podľa typu parametrov, s ktorými sa metóda volá.
2. **Prekonávanie metód:** Tiež využívam prekonávanie metód, kedy potomkova trieda poskytuje vlastnú implementáciu určitej metódy, ktorá je definovaná v nadradenej triede. Týmto spôsobom môžem meniť správanie určitých operácií v potomkových triedach.
3. **Použitie kľúčového slova super:** Využívam kľúčové slovo super, ktoré mi umožňuje volať konštruktor nadradenej triedy z vnútra potomkovej triedy. Týmto spôsobom môžem inicializovať vlastnosti, ktoré sú definované v nadradenej triede, a zároveň rozšíriť ich inicializáciu o ďalšie vlastné akcie.

Preťažovanie metód/Overloading + super

V triede Meeting som zadefinoval dva rôzne spôsoby vytváraná objektu pomocou konštruktorov a líšia sa vstupnými parametrami.

Rodičovská trieda
Inicializácia inštancie triedy pomocou Konštruktora 1

```
Meeting idealMEET = new Meeting(i, Integer.toString(counterID_meets), getCapacity(), getDuration());
```

```
1 usage  Martin Kvietok
public Meeting(int startTime, String roomID, int capacity, int duration) {
    super(capacity, duration);
    this.default_capacity = capacity;
    this.StartTime = startTime;
    this.RoomID = roomID;
    this.EndTime = startTime + duration;
}

3 usages  Martin Kvietok
public Meeting(int startTime, String roomID, int duration, int capacity, ArrayList<String> votes) {
    super(capacity, duration);
    this.StartTime = startTime;
    this.RoomID = roomID;
    this.EndTime = startTime + duration;
    this.Votes = votes;
}
```

Detská trieda
Inicializácia inštancie triedy pomocou Konštruktora 2

```
public OnlineMeet(int startTime, String roomID, int capacity, int duration, ArrayList<String> votes) {
    super(startTime, STR."\{roomID} online", capacity, duration, votes);
}
```

//-- Pomocou **super** dokáže trieda OnlineMeet komunikovať s jej nadtriednou Meeting
a využívať jej metódy a premenné.

Prekonávanie metód / overriding

Pri prekonaní metód sa mení alebo nahradzuje telo existujúcej metódy v nadradenej triede alebo implementuje metóda z rozhrania s vlastnou implementáciou v potomkovej triede. Týmto spôsobom potomkova trieda môže poskytnúť vlastnú verziu určitej metódy, ktorá sa líši od implementácie v nadradenej triede alebo z rozhrania.

Prekonanie pri implementácií rozhrania:
Trieda TimeTableBoard prekonáva metódy z implementovaného rozhrania TimeTable

```
3 usages  new *
@Override
public int getDuration() {
    return this.duration;
}

2 usages  Martin Kvietok
@Override
public int getCapacity() {
    return this.capacity;
}
2 usages  new *
@Override
public int getNumberOfMembers() {
    return users.size();
}

5 usages  new *
@Override
public void addUser(User user) {
    users.add(user);
}
```

Prekonanie rodičovskej metódy, prvá hierarchia dedenia:

Rodičovská metóda:

```
7 usages  1 override  Martin Kvietok *
@Override
public void vote(User user, TimeTableBoard Workspace) {
    Random random = new Random();
    int dayIndex = random.nextInt(Workspace.getNumOfDays());
    int timeIndex = random.nextInt(Workspace.getNumOfMeets());

    if (dayIndex < Workspace.getNumOfDays()) {
        Day day = Workspace.getDayByIndex(dayIndex);
        if (timeIndex < day.getNumOfMeets()) {
            Meeting meeting = day.getMeetingByIndex(timeIndex);
            boolean found = false;
            for (String userID : meeting.getvotedUsers()) {
                if (userID.equals(user.getUserID())) {
                    found = true;
                    break;
                }
            }
            if (!found) {
                meeting.add(user.getUserID());
                meeting.setCapacity(meeting.getCapacity() - 1);
                meeting.VoteForType();
                return;
            } else {
                vote( user: this, Workspace);
            }
        }
    }
}
```

```
2 usages  5 overrides  Martin Kvietok
@Override
public void setVote(TimeTableBoard Workspace) {
    Random random = new Random();
    int numberOfVotes = random.nextInt( bound: Workspace.getNumOfMeets() * Workspace.getNumOfDays()) + 1;
    for (int i = 0; i < numberOfVotes; i++) {
        vote( user: this, Workspace);
    }
    this.setHasVoted();
}
```

Detská metóda:

```
7 usages  Martin Kvietok
@Override
public void vote(User user, TimeTableBoard Workspace) {
    for (Day day : Workspace.getIdealDay()) {
        for (Meeting meeting : day.getIdealMEETs()) {
            meeting.add(this.getUserID());
            meeting.setCapacity(meeting.getCapacity() - 1);
            meeting.VoteForType();
        }
    }
}

2 usages  Martin Kvietok
@Override
public void setVote(TimeTableBoard Workspace) {
    vote( user: this, Workspace);
    this.setHasVoted();
}
```

//-- prekonanie metódy v kóde počas hlasovania

```
for (User user : visitedTown.getUsers()) {
    user.setVote(visitedTown);
}
```

Druhá hierarchia dedenia:

Rodičovská metóda:

```
3 usages  2 overrides  Martin Kvietok
public void setCapacity(int capacity) { this.capacity = capacity; }
```

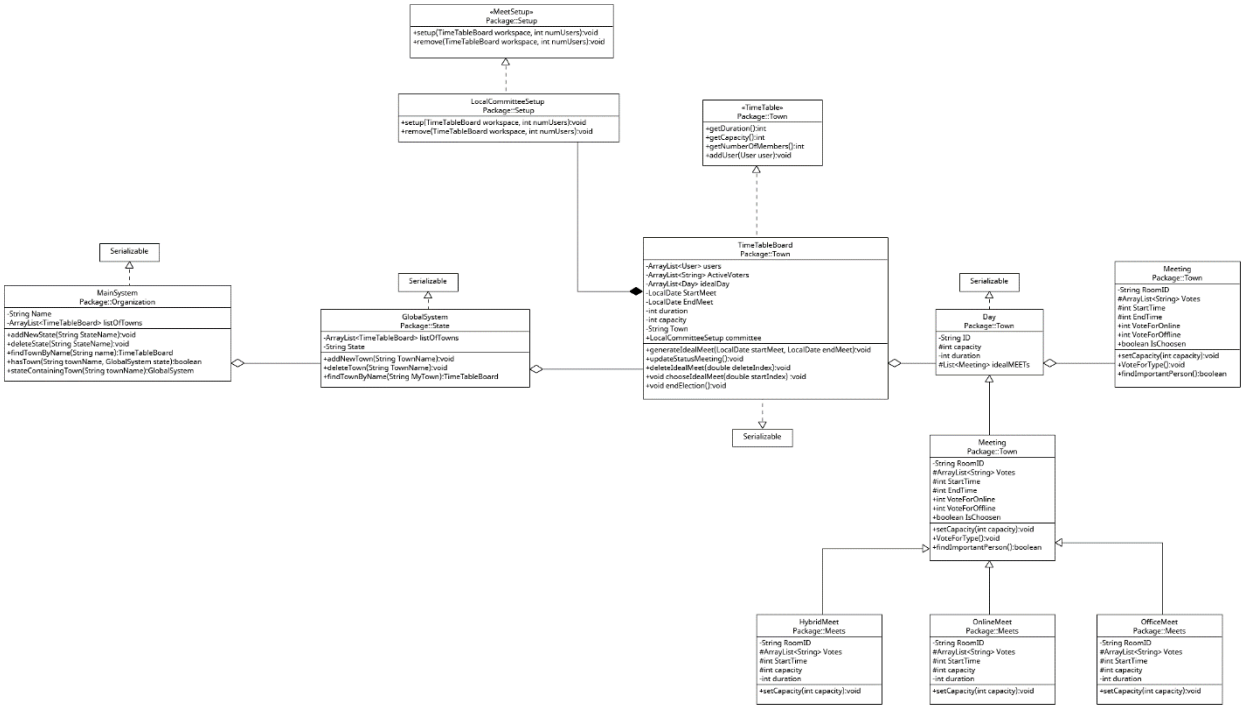
Detská metóda:

```
@Override
public void setCapacity(int capacity) {

    this.capacity = capacity;
    while (this.capacity < this.getNumberOfMembers())
    {
        this.capacity*=10;
    }
    this.EndTime = this.StartTime + this.capacity;
}
```

// prekonanie metódy v kóde počas hlasovania

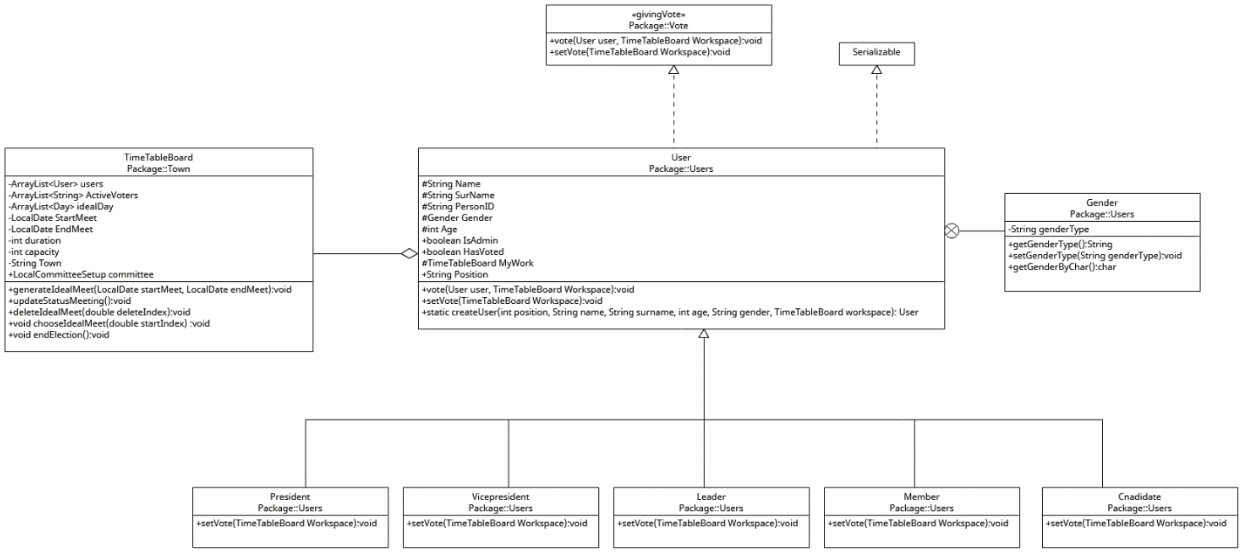
```
meeting.setCapacity(meeting.getCapacity());
```

Diagram tried



1. Vetva dedenia začína v triede `Day`, kde vidíme jednoduché dedičné vzťahy s triedou `Meeting`, ktorá slúži ako základná entita. Táto hierarchia sa ďalej vetví do tried `HybridMeet`, `OnlineMeet` a `OfficeMeet`, pričom v každej z týchto tried sa prekonáva metóda `void setCapacity(int capacity)`, ako sme už uviedli vyššie.

Trieda `MainSystem` reprezentuje celú organizáciu, ktorá agreguje pole triedy `GlobalSystem` (štáty). `GlobalSystem ďalej agreguje` pole tried `TimetableBoard` (mestá), v ktorých dochádza k hlasovaniu. Táto trieda implementuje rozhranie `TimeTable` so základnými metódami `set`/`get` a kompozitne vytvára triedu `LocalComitteSetup`, ktorá zabezpečuje princíp vytvárania komisií v každom meste (každé mesto má mať 1 prezidenta, 1 viceprezidenta + rozdelenie pomeru vodcov, členov a kandidátov). Taktiež implementuje rozhranie `MeetSetup`, kde prekonáva metódy na pridávanie a odstraňovanie používateľov z komisií. `Timetable agreguje` pole tried `Day` (dni konania mítingov), pričom každý deň má definované hodiny, kde prebiehajú jednotlivé stretnutia (`Meeting`).



2. hierarchia dedičnosti začína v triede `User`. Dedičné vetvy sú definované podľa postavenia osoby v organizácii, ako sú `President`, `Vicepresident`, `Leader`, `Member` a `Candidate`. Každé z týchto postavení má odlišný počet hlasov, ktorý je možné nastaviť pomocou prekonania metódy `setVote`, implementovanej z rozhrania `givingVote`.

Okrem toho trieda `User` obsahuje vnútornú triedu `Gender`, ktorá obsahuje premennú a funkciu na nastavenie a získanie pohlavia danej osoby.

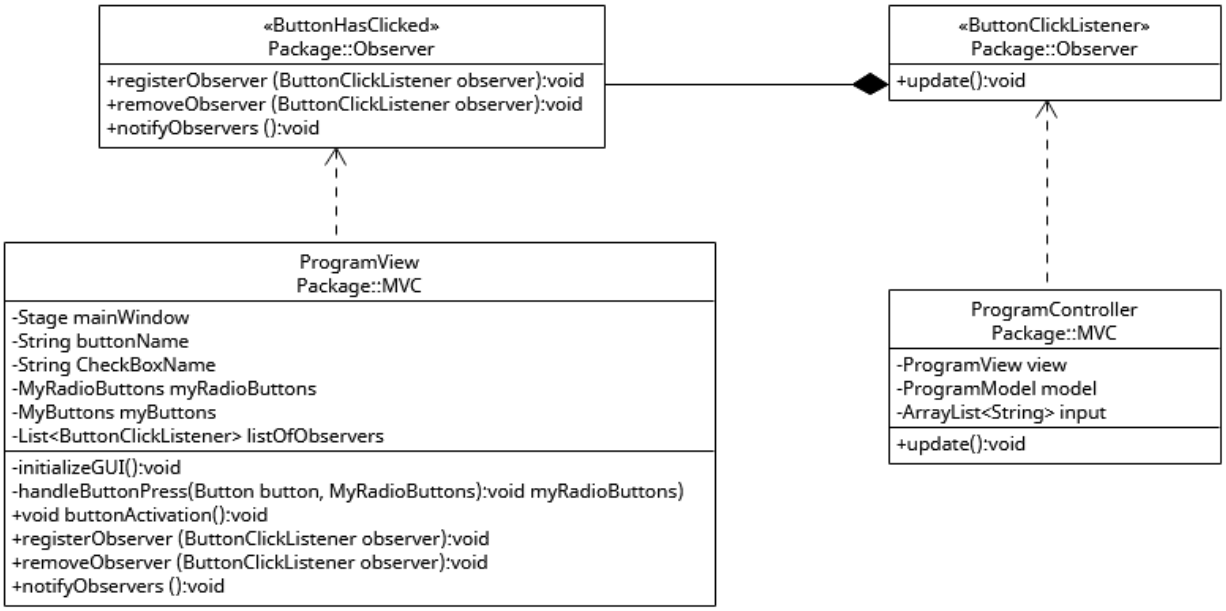
Každý používateľ je ďalej pridávaný do poľa tried v `TimetableBoard`, teda v meste, v ktorom pôsobí.

Ďalšie kritériá

Návrhový vzor Observer

V mojom projekte som použil návrhový vzor Observer na sledovanie stlačenia tlačidla v používateľskom rozhraní (GUI). Tento vzor mi pomohol efektívne komunikovať medzi rôznymi komponentami v architektúre MVC.

Medzi Subjectom a Observerom existuje kompozičný vzťah, pretože každý z nich potrebuje existenciu druhého na to, aby mohol správne fungovať.



V mojom prípade má trieda ButtonHasClicked úlohu "Subjectu", ktorý oznamuje zmenu svojho stavu.

```
package GUI;

import ...

2 usages  2 implementations  Martin Kvietok *
public interface ButtonHasClicked {
    2 usages  2 implementations  new *
    public void registerObserver (ButtonClickListener observer);
    no usages  2 implementations  new *
    public void removeObserver (ButtonClickListener observer);
    4 usages  2 implementations  new *
    public void notifyObservers () throws DuplicateError, NotFound, WrongDateException;
}
```

Na druhej strane, trieda ButtonClickListener zohráva úlohu "Observera", ktorý sleduje zmenu stavu ButtonHasClicked.

```
package GUI;

import ...

14 usages  2 implementations  Martin Kvietok *
public interface ButtonClickListener {
    //-- observer pattern
    2 usages  2 implementations  Martin Kvietok
    public void update() throws DuplicateError, NotFound, WrongDateException;
}
```

Trieda ProgramView je subjektom sledovania, preto si udržiava zoznam svojich pozorovateľov pomocou funkcií ako register, remove a notify, ktoré sú prebrané zo subjektu.

```
view.registerObserver(controller);
```

```
3 usages
private List<ButtonClickListener> listOfObservers = new ArrayList<~>();
```

```
// --- observer pattern

2 usages  Martin Kvietok
@Override
public void registerObserver(ButtonClickListener observer) { listOfObservers.add(observer); }

no usages  Martin Kvietok
@Override
public void removeObserver(ButtonClickListener observer) { listOfObservers.remove(observer); }

4 usages  Martin Kvietok
@Override
public void notifyObservers() throws DuplicateError, NotFound, WrongDateException {
    for (ButtonClickListener buttonClickListener : listOfObservers) {
        buttonClickListener.update();

    }
}
```

```
public void buttonActivation() {
    for (Button button : systemButtons) {
        button.setOnAction(event -> {
            handleButtonPress(button, myRadioButtons);
            try {
                notifyObservers();
            } catch (DuplicateError | NotFound | WrongDateException e) {
                throw new RuntimeException(e);
            }
        });
    }
```

Trieda ProgramController pozoruje iné triedy (v tomto prípade iba ProgramView) a implementuje rozhranie Observer s operáciou na aktualizáciu svojho stavu podľa sledovanej triedy.

```
@Override
public void update() throws DuplicateError, NotFound, WrongDateException {
    switch (view.getButtonName()) {

        case "X":
            model.serializeObject();
            break;

        case "Create Organization":
            view.disableKeyButtons();
            view.selectOrganizationGUI();
            numInput = 1;
            indexInput = 0;
            break;

        case "State":
            view.disableKeyButtons();
            view.selectStateGUI();
            numInput = 1;
            indexInput = 1;
            break;
```

Ošetrenie mimoriadneho stavu pomocou vlastnej výnimky – throw & handle

V mojom programe som implementoval 3 vlastné výnimky, ktoré slúžia na ošetrenie mimoriadnych situácií:

1. **WrongDateException** - Táto výnimka sa používa v prípade, že používateľ zvolí dátum pre meeting, ktorý už uplynul. Je to dôležité pre zabezpečenie, aby sa v programe pracovalo iba s aktuálnymi alebo budúcimi udalosťami.
2. **DuplicateError** - Táto výnimka sa vyvolá, keď sa pokúšame pridať nový prvok do Listu a zistíme, že tento prvok už v danom Liste existuje. Pomáha to udržať konzistenciu dát a zabrániť duplikátom.
3. **NotFound** - Táto výnimka je užitočná v situáciách, keď sa snažíme pracovať s prvkom, ktorý neexistuje v danom Liste. To zabezpečuje bezpečné správanie programu pri manipulácii s dátami.

Všetky tieto výnimky boli implementované s cieľom zabezpečiť robustnosť a bezpečnosť kódu. Ich správne použitie prispieva k správnemu fungovaniu programu a pomáha predchádzať chybám a nekonzistentným stavom.

Definovanie vlastnej podmienky WrongDateException

```
20 usages  Martin Kvietok
public class WrongDateException extends Exception {
    2 usages
    private String errorMessage;

    1 usage  Martin Kvietok
    public WrongDateException(String message) { this.errorMessage = message; }

    Martin Kvietok
    @Override
    public String toString() { return STR."WrongDateError: \{errorMessage}"; }
}
```

Následne v triede TimeTableBoard, v metóde setDates(), som kontroloval správnosť zadaných dátumov nasledovným spôsobom:

```
1 usage   Martin Kvietok
public void setDates(String Start, String End) throws WrongDateException {
    LocalDate currentDate = LocalDate.now();

    this.StartMeet = LocalDate.parse(Start, formatter);
    this.EndMeet = LocalDate.parse(End, formatter);

    if (StartMeet.isBefore(currentDate) || EndMeet.isBefore(currentDate)) {
        throw new WrongDateException("Wrong date insert.");
    }
}
```

V prípade zlého vstupu bola podmienka odchytená v metóde createMeeting(), ktorá zapríčinila aj adekvátny výpis chybovej hlášky do textového poľa na obrazovke.

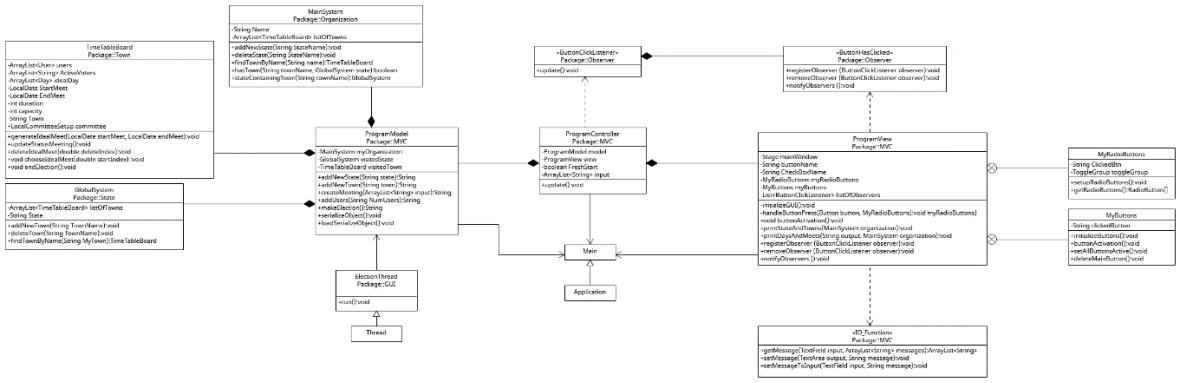
```
1 usage   Martin Kvietok
public String createMeeting(ArrayList<String> input) throws WrongDateException {
    visitedTown = myOrganization.findTownByName(input.get(0));
    if (visitedTown != null) {
        if (!visitedTown.getIdealDay().isEmpty()) {
            return "iMeet has been already set.\n-----\n";
        }
        try {
            visitedTown.setupTimeTableB(input.get(1), input.get(2), Integer.parseInt(input.get(3)), Integer.parseInt(input.get(4)));
            return STR."iMeet in \{visitedTown.getTown()} was successfully created\n-----\n";
        } catch (WrongDateException wrongDateException) {
            return wrongDateException.toString();
        }
    } else {
        return STR."\{input.getFirst()} not found\n-----\n";
    }
}
```

Oddelenie GUI od aplikačnej logiky

Tento problém som vyriešil implementovaním architektonického vzoru MVC (Model-View-Controller). V tejto architektúre:

- **Model** reprezentuje dáta organizácie alebo aplikácie. Obsahuje logiku a funkcionalitu potrebnú na prácu s týmito dátami.
- **Controller** je zodpovedný za správu toku dát v aplikácii a manipuláciu s nimi. Zabezpečuje vstupy a výstupy a aktualizuje View, keď je to potrebné.
- **View** sa stará o vizualizáciu dát. Je to časť aplikácie, ktorú používatelia vidia na obrazovke alebo na výstupe.

V rámci tejto architektúry Model obsahuje samotné dáta, Controller spravuje interakcie s týmito dátami a aktualizuje View podľa potreby, aby používateľ videl aktuálny stav dát. Týmto spôsobom sa oddeľuje logika od prezentačnej vrstvy a dosahuje sa lepšia organizácia a údržba kódu.



GUI s event handler-mi

Vytvoril som užívateľské rozhranie pomocou JavaFX, ktoré umožňuje interakciu s rôznymi hlavnými tlačidlami checkboxami. Hlavné tlačidlá slúžia na vykonávanie kľúčových operácií, ako je vytvorenie organizácie, štátu, mesta, stretnutia, používateľa, hlasovanie alebo hromadného hlasovania.

Checkboxy, konkrétne možnosti Pridať, Odstrániť a Žiadne, sú určené na výber možností alebo nastavení, ktoré ovplyvňujú prácu s dátami.

Na obrazovke sú umiestnené tri výstupné okná, kde sa dynamicky zobrazuje aktuálny stav organizácie, ako je počet miest, používateľov a výsledky volieb. Tieto informácie sa pravidelne aktualizujú, aby používateľ videl najnovšie údaje.

Umožňujem aj vstup textu, ktorý je potrebné potvrdiť stlačením tlačidla Enter. To poskytuje flexibilitu pri zadávaní údajov a interakcii s aplikáciou.

```
//-- event handlers
1 usage   Martin Kvietok
public void buttonActivation() {
    for (Button button : systemButtons) {
        button.setOnAction(event -> {
            handleButtonPress(button, myRadioButtons);
            try {
                notifyObservers();
            } catch (DuplicateError | NotFound | WrongDateException e) {
                throw new RuntimeException(e);
            }
        });
    }
}
```

V príklade som tieto tlačidlá a kontrolky typu checkbox použil na notifikáciu observera o tom, že bolo tlačidlo stlačené, a následne ukladám jeho hodnotu.

```
    }
    enterButton.setOnAction(event -> {
        handleButtonPress(enterButton, myRadioButtons);
        try {
            notifyObservers();
        } catch (DuplicateError | NotFound | WrongDateException e) {
            throw new RuntimeException(e);
        }
    });
    mainStage.setOnCloseRequest(event -> {
        buttonName = "X";
        try {
            notifyObservers();
        } catch (DuplicateError | NotFound | WrongDateException e) {
            throw new RuntimeException(e);
        }
        System.out.println("Window is closing...");
    });
```

Multithreading

Vytvoril som vlastnú triedu ElectionThread, ktorá slúži ako vlákno pre procesy súvisiace s voľbami, kde všetci účastníci volia naraz. Pre každé aktívne mesto sa vytvára nové vlákno a pridáva sa do zoznamu. Po tom, ako sa vlákno vytvorí, je zavolaná metóda start(), ktorá spúšťa nové vlákno a volá jeho metódu run().

Na zabezpečenie synchronizácie a správneho správania programu som využil metódu join(). Táto metóda zabezpečuje, že hlavné vlákno programu bude čakať, kým všetky volebné vlákna vytvorené v priebehu iterácie cyklu budú ukončené, predtým ako sa pokračuje ďalšou časťou kódu. Týmto spôsobom sa zabezpečuje, že všetky významné časti procesu volieb sú dokončené pred ďalším vykonávaním programu.

```java
public class ElectionThread extends Thread {
    2 usages
    private TimeTableBoard town;

    1 usage  Martin Kvietok
    public ElectionThread(TimeTableBoard town) {
        this.town = town;
    }

    Martin Kvietok
    public void run() { makeElection(town); }

    1 usage  Martin Kvietok *
    private void makeElection(TimeTableBoard visitedTown) {
        for (User user : visitedTown.getUsers()) {
            user.setVote(visitedTown);
        }
        double index = 0.5;
        while (visitedTown.getNumOfMeets() >= 3) {
            visitedTown.chooseIdealMeet(index);
            index /= 2;
        }
    }
}
```

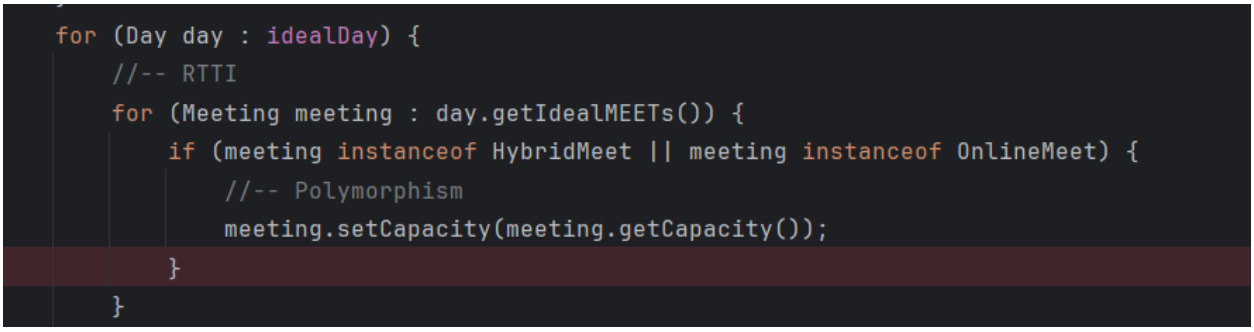
```java
//-- multithreading
1 usage  Martin Kvietok
public void electionForAllTowns() {
    List<Thread> threads = new ArrayList<>();

    for (GlobalSystem state : myOrganization.getListOfStates()) {
        for (TimeTableBoard town : state.getListOfTowns()) {
            if (!town.getIdealDay().isEmpty())
            {
                ElectionThread electionThread = new ElectionThread(town);
                threads.add(electionThread);
                electionThread.start();
            }
        }
    }

    for (Thread thread : threads) {
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

RTTI – Runtime Type Identification

Využívam RTTI (Run-Time Type Identification) cez operátor instanceof. V triede TimeTableBoard, konkrétne v metóde updateStatusMeeting(), používam tento mechanizmus na identifikáciu toho, či je daný objekt typu Meeting inštanciou HybridMeet alebo OnlineMeet. To je potrebné pre zvýšenie jeho kapacity.

Pomocou instanceof môžem dynamicky určiť typ objektu za behu programu a podľa toho prijať rozhodnutie. V tomto prípade, ak je objekt typu HybridMeet alebo OnlineMeet, zvýšim jeho kapacitu.

Tento prístup mi umožňuje flexibilne pracovať s rôznymi typmi objektov v závislosti od ich aktuálneho typu za behu programu, čo môže byť užitočné pre rôzne operácie a manipulácie s dátami.



Vhniezdené triedy

V mojom projekte som implementoval jednu vnorenú triedu mimo používateľského rozhrania (GUI) v rámci triedy User. Vnorená trieda Gender je využívaná na vykonávanie a uchovávanie informácií o pohlaví používateľa. Táto vnorená trieda nahradzuje funkcionalitu dátového typu ENUM.

Použitie vnorenej triedy umožňuje usporiadať kód a skupinovať súvisiace triedy spolu. Týmto spôsobom je logika týkajúca sa pohlavia používateľa oddelená od zvyšku kódu a zjednodušuje sa správa a údržba projektu. Vnorená trieda Gender poskytuje flexibilný a prehľadný spôsob reprezentácie pohlavia používateľa v rámci triedy User.

```
//-- nested class
3 usages  Martin Kvietok
public static class Gender implements Serializable {
    4 usages
    private String genderType;

    1 usage  Martin Kvietok
    public Gender(String genderType) { this.genderType = genderType; }

    no usages  Martin Kvietok
    public String getGenderType() { return genderType; }

    no usages  Martin Kvietok
    public void setGenderType(String genderType) { this.genderType = genderType; }

    1 usage  Martin Kvietok
    public char getGenderByChar() {
        if (genderType.equals("MALE")) {
            return 'X';
        } else {
            return 'Y';
        }
    }
}
```

Default method

V mojom projekte som implementoval metódu DEFAULT v rozhraní IO_function, ktoré implementuje trieda ProgramView. Tento prístup je efektívne riešenie pre správu a manipuláciu so vstupno-výstupnými informáciami.

Metóda DEFAULT v rozhraní IO_function definuje predvolenú implementáciu určitých operácií týkajúcich sa vstupu a výstupu. Táto implementácia poskytuje základnú funkcionalitu, ktorá môže byť v triede ProgramView priamo použitá alebo ďalej rozšírená podľa potreby.

```
1 usage  1 implementation  Martin Kvietok
interface IO_Function {

    //-- default method implementation
    1 usage  Martin Kvietok
    default ArrayList<String> getMessage(TextField input, ArrayList<String> messages) {
        messages.add(input.getText());
        input.clear();
        return messages;
    }

    30 usages  Martin Kvietok
    default void setMessage(TextArea output, String message) { output.appendText(message); }

    8 usages  Martin Kvietok
    default void setMessageToInput(TextField input, String message) { input.setPromptText(message); }
}
```

```
public ArrayList<String> scanInput() { return this.getMessage(Input, InputArray); }
```

```
12 usages  Martin Kvietok
public void report(String message) { this.setMessage(generalOutput, message); }
```

```
1 usage  Martin Kvietok
public void selectOrganizationGUI() {
    this.setMessage(generalOutput,  message: "Please enter your organization name:\n");
    this.setMessageToInput(Input,  message: "organization name");
}
```

Serializácia

V mojom programe serializácia sa spúšťa vždy pri ukončení programu. Serializujem dátovú triedu MainSystem v rámci ProgramModelu, ktorá obsahuje všetky informácie o organizácii, ako sú štáty, mestá, používateľov a vytvorené stretnutia. Na začiatku programu sa používateľovi vždy ponúka možnosť vybrať, či chce pokračovať (a teda deserializovať objekt) alebo začať úplne od začiatku.

Tento prístup umožňuje užívateľovi pokračovať v práci tam, kde skončil, ak si to želá, a zároveň poskytuje možnosť začať novú prácu od nuly, ak je to potrebné alebo žiadané. Serializácia a deserializácia teda pomáhajú uchovávať a obnovovať stav programu medzi rôznymi spusteniami.

```
    //-- object serialization
    1 usage  Martin Kvietok
    public void serializeObject() {
        try (FileOutputStream fos = new FileOutputStream( name: "iMeet.ser");
             ObjectOutputStream oos = new ObjectOutputStream(fos)) {
            oos.writeObject(myOrganization);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    1 usage  Martin Kvietok
    public void loadSerializeObject() {
        try (FileInputStream fis = new FileInputStream( name: "iMeet.ser");
             ObjectInputStream ois = new ObjectInputStream(fis)) {
            myOrganization = (MainSystem) ois.readObject();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

}
```

```
    if (view.getButtonName().equals("Continue")) {
        model.loadSerializeObject();
        this.FreshStart = false;
        showMainWindow();
    } else {
        showMainWindow();
    }
}
```

```
case "X":
    model.serializeObject();
    break;
```

```
9 usages  Martin Kvietok *
public class MainSystem implements Serializable {
    2 usages
    protected String name;
```

Splnenie kritérií

Hlavné kritéria

Čo sa týka hlavných kritérií, projekt **plní všetky** hlavné kritéria pracovnej verzie – adekvátne použité dedenie, polymorfizmus, agregácia, GUI oddelené od aplikačnej logiky.
Projekt **plní všetky** hlavné kritéria finálnej verzie – program je funkčný, zodpovedá zadaniu spresnenému v odovzdanom zámere projektu, program obsahuje zmysluplné dedenie s prekonávaním vlastných metód v aspoň 2 prípadoch, v programe je použitá enkapsulácia, takisto implementácia rozhrania, kód obsahuje dostatok komentáru pre správne pochopenie mojich myšlienkových pochodov, dokumentácia zodpovedá programu a obsahuje diagram tried so všetkými triedami a vysvetlením najdôležitejších vzťahov.

Vedľajšie kritériá

Program obsahuje nasledovné ďalšie (vedľajšie) kritériá (všetky zahrnuté v tejto dokumentácii):

1. použitie návrhových vzorov – Observer
2. ošetrenie mimoriadnych stavov prostredníctvom vlastných výnimiek
3. poskytnutie grafického používateľského rozhrania oddelene od aplikačnej logiky a s aspoň časťou spracovateľov udalostí
4. explicitné použitie viacniťovosti (multithreading)
6. explicitné použitie RTTI – na zistenie typu objektu
7. použitie vhniezdených tried a rozhraní
9. použitie implicitnej implementácie metód v rozhraniach (default method)
11. použitie serializácie

Zoznam hlavných verzií programu

My First Commit (a2ef02b4b8ab48d170e02b60e267fd460d5be6dc)

- Vytvorenie základného návrhu a hlavných tried hierarchie (MainSystem,GlobalSystem,TimeTableBoard, Meeting)

adding more users (42c1067cc495fd999b00300b47bf2408f44ef534)

- Pridanie ďalšej hierarchie a tried zámernej na vetvu užívateľov User (President,Vice,Leader,Member,Candidate)

MVC pattern (8b3bc11e987598bfb572f20f87a0d8eedd39a265) ver.1

- Pokus o oddelenie aplikačnej logiky od používateľského rozhrania (nedostatočný – všetko v jednej triede)
- JavaFX
- Fungujúce hlasovanie

MVC pattern (b413a9c9c4cf91c6b8b2ebc6be4854a385e8cbe8) ver.2

- Úspešné oddelenie aplikačnej logiky od používateľského rozhrania (obsahuje samostatné triedy ProgramView,ProgramController,ProgramModel)

MVC pattern2 (2afcc12d3652eccebe69b330999aa246a433fab3)

- Pridanie fungujúceho Observera a ostatných kritérií
- Pridanie vstupného okna

Verzie vyššie obsahujú len opravy a refaktorizáciu kódu + informácie pre Javadoc